ソフトウェア実験Ⅱ・Ⅲ
テーマ１ レポート

学籍番号：1TE22028G

氏名：一瀬遥希

提出日：2024 年 12 月 13 日

1　テーマ 1 についての説明

テーマ 1 では, C 言語とアセンブリ言語を組み合わせて, C 言語処理系 m68kelf-gcc に付随して提供されているライブラリのうち, 基本的な関数である read(), write(), printf(), scanf()が MC68VZ328 ボードにおいて動作する環境を構築することを目的としている. これにより, システムレベルでのプログラム操作や, 標準入出力を通じたユーザーインターフェースの基本を学ぶことができる.

具体的には, シリアルポートを使用してデータの入出力を行うための inbyte と outbyte という 2 つのアセンブリ言語サブルーチンを実装する. このサブルーチンは, モニタプログラムが提供するシステムコール GETSTRING および PUTSTRING を利用することで, 1 文字単位の入出力を行う. これらの実装を通じて, C 言語からアセンブリ言語を呼び出す方法や, 引数や戻り値の取り扱い方法を学ぶ.

このテーマ 1 は, C 言語プログラムがアセンブリ言語の低レベル操作を活用する際の基本的な技術を習得する重要なステップとなる. 特に, テーマ 2 以降のマルチタスクカーネル開発や応用プログラム作成の準備段階として位置付けられる.

2　テーマ 1 の実験全体に占める位置付けの説明

テーマ 1 は実験全体の基盤を形成する重要な役割を持つ. C 言語とアセンブリ言語間の連携方法を学ぶことで, テーマ 2 以降のマルチタスクカーネルや応用プログラムの作成に必要な知識を習得できる.

例えば, テーマ 2 では複数タスクの切り替え処理やリソース管理を扱い, テーマ 3 では実際の応用プログラムを開発するが, それらすべてにおいて C 言語とアセンブリ言語の連携が必須となる. 特に, テーマ 1 で実装する inbyte と outbyte を基に, 標準入出力ライブラリの機能を利用する read や write を動作させることが可能になる. このため、テーマ 1 は以降のテーマで必要となる基礎スキルを身に付けるための準備段階として非常に重要である.

3　プログラムのリスト・説明

3.1　モニタプログラムの初期化

以下に, モニタプログラムの初期化部分のプログラムリストを示し, 今回の実験のための変更点について説明を行う.

```
.extern     start
.global     monitor_begin
.include    "equdefs.inc"

monitor_begin:

.section .bss
.even
SYS_STK: .ds.b 0x4000    | システムスタック領域

.even
SYS_STK_TOP:         | システムスタック領域の最後尾

.even
task_p: .ds.l 1

*****************************************************************
** 初期化
*****************************************************************
.section .text
.even


boot:   ************************************************
    ** スーパーバイザ & 各種設定を行っている最中の割込禁止
    ************************************************
    move.w  #0x2700,%SR
    lea.l   SYS_STK_TOP, %SP | Set SSP

    ******************************
    ** 割り込みコントローラの初期化
    ******************************
    move.b  #0x40, IVR | ユーザ割り込みベクタ番号を 0x40+level に設定.
    move.l  #0x00ff3ffb,IMR | 全割り込みマスク

    *****************************************************************
```

```
** 送受信（UART1）関係の初期化（割り込みレベルは 4 に固定されている）
*****************************************************************
move.w  #0xe10c, USTCNT1
move.w  #0x0038, UBAUD1 | baud rate = 230400 bps

*********************************************************
** タイマ関係の初期化（割り込みレベルは 6 に固定されている）
*********************************************************
move.w  #0x0004, TCTL1

move.l  #uart1_interrupt, 0x110
move.l  #timer_interrupt, 0x118

***************
**キューの初期化
***************
jsr     Init_Q

move.l  #SYSTEM_CALL,0x080
move.l  #0x00f3ff9, IMR



move.w  #0x0000, %SR
jmp start
```

C 言語プログラムは，最初に main 関数が実行されるが，その前にスタックポインタや BSS セクションの初期化などが行われる．本実験では，前回作成したモニタプログラムの初期化ルーチンを活用し，以下のような変更を加えることで，C プログラムの main 関数を起動するようにした．

1. 初期化ルーチンの先頭に monitor_begin というラベルを追加
2. 初期化ルーチン終了後に、main 関数がある start にジャンプするための jmp start を記述
3. 他のファイルから monitor_begin を呼び出せるように.global monitor_begin を宣言し，start を外部参照するために.extern start を宣言

これにより, C プログラムの実行前に必要な初期化処理をモニタプログラムで行うことが可能になった.

3.2 inbyte()のアセンブリ言語による作成

3.2.1 inbyte()を含む inchrw.s

以下に, inbyte サブルーチンを含む inchrw.s ファイル内のプログラムリストとその説明を示す.

```
.global inbyte
.include "equdefs.inc"

.text
.even

inbyte:
    movem.l %d1-%d3, -(%sp)

inbyte_loop:

    move.l  #SYSCALL_NUM_GETSTRING, %d0
    move.l  #0, %d1
    move.l  #inchrw_buf, %d2
    move.l  #1, %d3
    trap    #0

    cmpi.l   #0, %d0
    beq     inbyte_loop

    move.b  inchrw_buf, %d0
    move.b  %d0, LED0

    movem.l (%sp)+, %d1-%d3
    rts

.section .bss
.even
```

```
inchrw_buf: .ds.b 1
    .even
```

inbyte()及び inbyte_loop()は，引数を取らず，戻り値としてシリアルポート 0 から読み込んだ 1 文字(char 型)を返すアセンブリ言語サブルーチンである．この関数は，モニタプログラムが提供する GETSTRING システムコールを使用する．

関数の構成は以下の通りである．

1. レジスタ退避

   サブルーチンの先頭で%d1 から%d3 のレジスタをスタックに退避し，サブルーチン内での変更が外部に影響しないようにしている．

2. システムコール呼び出し準備

   GETSTRING システムコール番号を%d0 に設定する．入力する文字を格納するメモリ上のバッファ（inbyte_buff）アドレスを%d2 に設定する．データのサイズは%d3 に 1 バイトと指定する．

3. システムコール実行

   trap #0 命令でシステムコールを呼び出す．この時点でシステムはシリアルポート 0 から 1 文字のデータを受信し，指定されたバッファに格納する．

4. 入力の確認とループ処理

   GETSTRING が成功すると，戻り値として%d0 に 1 が格納される．もし戻り値が 1 でなければ，入力が正しく行われなかったことを意味するため，再度ループしてシステムコールを呼び出す．

5. 戻り値設定

   入力された文字は inbyte_buff に格納されているため，それを%d0 に移動し，C 言語から利用可能な形にする．

6. レジスタ復元

   最後にスタックから%d1 から%d3 を復元し，サブルーチンを終了する．

このサブルーチンにより，確実に 1 文字を入力し，その結果を C 言語の関数から返り値として利用できるようになっている．ここでは，プログラム内で 1 バイトサイズでバッファを定義し，それを入力を受け取るために利用している．

3.2.2　Option: 非待機 1 文字入力 inkey()

以下に, Option として上記の inchrw.s 内に追加作成を行った inkey()のプログラムリストとその説明を示す.

```
inkey:
    movem.l %d1-%d3, -(%sp)

inkey_loop:
    move.l  %d0, %d1
    move.l  #SYSCALL_NUM_GETSTRING, %d0
    move.l  #inchrw_buf, %d2
        move.l  #1, %d3
        trap    #0

        cmpi.l   #0, %d0
        beq     no_input

        move.b  inchrw_buf, %d0
    andi.l  #0xff, %d0

no_input:
    move.l  #-1, %d0

input_end:
    movem.l (%sp)+, %d1-%d3
    rts
```

この関数は, inbyte()と同様の構成ではあるが, 戻り値を 1 文字入力がある場合はその文字を符号なし int 型に変換した値とし, 1 文字入力がない場合は-1(0xff)となるように記述したものである. なおこの関数は動作の確認は取っておらず, テーマ 3 の応用プログラムのための準備として作成を行った.

3.2.3　中難度 Option: C のローカル変数領域に相当する作業領域の確保

以下に, 中難度 Option 課題としてプログラム内でブッファを定義するのではなくローカル変数を作業領域として取った inbyte()を含む inchrw.s のプログラムリストと説明を示す.

```
.global inbyte
.include "equdefs.inc"

.text
.even

inbyte:
    movem.l %d1-%d3/%a6, -(%sp)
    link.w  %a6, #-4

inbyte_loop:
    move.l  #SYSCALL_NUM_GETSTRING, %d0
    move.l  #0, %d1
    move.l  %a6, %d2
    sub.l   #1,  %d2
    move.l  #1, %d3
    trap    #0

    cmpi.l   #0, %d0
    beq     inbyte_loop

    move.b  -1(%a6), %d0
    move.b  %d0, LED0
    unlk    %a6
    movem.l (%sp)+, %d1-%d3/%a6

    rts
```

ここでの inbyte()及び inbyte_loop()は,C のローカル変数領域に相当する作業領域を確保し、効率的なメモリ操作を実現している.

動作手順は以下の通りである.

1. レジスタの退避

関数の先頭で,データレジスタ%d1～%d3 とアドレスレジスタ%a6 をスタックに退避する。これにより,サブルーチン内でこれらのレジスタが変更された際の,関数外部への

影響を防いでいる．

2. ローカル作業領域の確保

link.w %a6, #-4 命令を使用して，スタックフレームを確保し，作業領域を設定する．この作業領域を通じてデータの一時的な格納が可能となる．

3. 入力ループの開始

システムコール番号 SYSCALL_NUM_GETSTRING を%d0 に，バッファアドレスを%d2 に，データサイズ（1 バイト）を%d3 に設定する．そして trap #0 命令を実行してシステムコールを呼び出し，シリアルポート 0 からデータを受信する．

4. 入力成功の判定

戻り値として%d0 に成功フラグ（1 が成功、0 が失敗）が格納ため，cmpi.l #0, %d0 命令で失敗を判定し，成功するまでループを繰り返す．

5. データの取得と格納

入力された文字は作業領域内のバッファ（-1(%a6)）に格納されている．この値を%d0 に移動し，関数の戻り値として設定する．また，入力された文字を LED0 に出力することで，デバッグや動作確認が行っている．

6. 作業領域の解放

unlk %a6 命令を使用してスタックフレームを解放し，作業領域を元の状態に戻す．

7. レジスタの復元と終了

スタックから退避していたレジスタを復元し，rts 命令で関数を終了する．

このような手順で，ローカル作業領域の確保と LED を用いた簡易デバッグ機能を組み合わせて，より実用的な設計とした．また，入力データの成功確認をループで行うことで，確実なデータ取得を保証している．

3.3 outbyte()のアセンブリ言語での作成

以下に，outbyte サブルーチンを含むファイル内のプログラムリストとその説明を示す．

```
.global outbyte
.include "equdefs.inc"
```

```
.text
.even

outbyte:
    movem.l %d1-%d3, -(%sp)

outbyte_loop:
    move.l  #SYSCALL_NUM_PUTSTRING, %d0
    move.l  #0, %d1
    move.l  %sp, %d2
    addi.l  #19, %d2
    move.l  #1, %d3
    trap    #0

    cmpi.l   #0, %d0
    beq outbyte_loop

    movem.l  (%sp)+, %d1-%d3
    rts
```

outbyte は，引数として char 型のデータを取り，シリアルポート 0 に出力するためのアセンブリ言語のサブルーチンである．この関数は，モニタプログラムが提供する PUTSTRING システムコールを使用する．

動作手順は以下の通りである．

1. レジスタ退避

   サブルーチンの先頭で%d1 から%d3 のレジスタをスタックに退避し，サブルーチン内での変更が外部に影響しないようにしている．

2. 引数の取り出し

   C 言語側から渡された出力文字は、スタックに保存されている．スタックポインタ（%sp）から 19 バイト進んだ位置に出力文字が格納されているため，アドレス計算を行い，出力文字のアドレスを%a0 に設定する．

3. システムコール呼び出し準備

PUTSTRING システムコール番号を%d0 に設定する．出力するデータのアドレスは%d2 に設定する．データのサイズは%d3 に 1 バイトと指定する．

4. システムコール実行

trap #0 命令でシステムコールを呼び出す．この時点で指定されたデータがシリアルポート 0 を通じて出力される．

5. 出力の確認とループ処理

PUTSTRING が成功すると，戻り値として%d0 に 1 が格納される．もし戻り値が 1 でなければ，出力が正しく行われなかったことを意味するため，再度ループしてシステムコールを呼び出す．

6. レジスタ復元

最後にスタックから%d0 から%d3 を復元し，サブルーチンを終了する．

このサブルーチンにより，C 言語のプログラムから引数で指定された文字を確実に出力することができる．

3.4 テスト用プログラム test1.c

3.4.1 test1.c

以下に，テスト用の C プログラムである test1.c のプログラムリストとその説明を示す．

```c
#include <stdio.h>

int main()
{

  while(1){
    char str[256];

    scanf("%s",str);
    printf("%s¥n",str);
    }
    return 0;
}
```

このプログラムは非常に簡素なものであり，入力した文字をそのまま出力する scanf 関数と Enter キーを押すことで入力された文字を表示する printf 関数を無限ループで繰り返すという構造になっている．このプログラムが正常に動作したため，作成した inbyte と outbyte によって，標準入出力関数が正常に動作することが確認できた．

3.4.2　低難度 Option: ユーザ定義 exit()による main()終了時のプログラム停止措置

以下に，同じソースモジュール内に関数 exit()を定義した test1.c のプログラムリストとその説明を示す．

```
#include <stdio.h>

int main()
{
  char str[256];

    scanf("%s",str);
    printf("%s¥n",str);

    return 0;
}
void exit(int value) {
*(char *)0x00d00039 = 'H'; /* LED0 への表示 (HALT) */
for (;;) ; /* 無限ループトラップで停止させる */
}
```

本実験における実機環境では main()の終了後は特に考慮されていないため，main()のテストプログラムはループ内記述が推奨されている．ここでは資料に従い，万が一 main()が終了してしまう場合を想定して，PC が text 領域外に置かれて誤動作を引き起こさないように，exit()を追加定義している．実際に，上記のプログラムリストにあるように，main()内のループ処理を外し，終了するようにして実行すると，LED0 へ”H”を表示してプログラムは正常に停止した．

3.5　低難度 Option: trap #0 のシステムコール番号 5 の追加

以下に，システムコール番号 5 の追加を行ったモニタプログラム mon.s の該当部分のプログラムリストを示し、説明を行う．

```
**************************
** SYSTEM_CALL
** 入力
** d0:システムコール番号
** d1 以降:システムコールの引数
** 出力
** d0:システムコール呼び出しの結果
**************************

SYSTEM_CALL:
    cmp.l   #5, %d0
    bne     SYSTEM_CALL1
    jsr     skipmt

SYSTEM_CALL1:
    movem.l %d1-%d3,-(%sp)      /*レジスタの退避*/
    cmp.l   #1,%d0
    bne SYSTEM_CALL2        /*システムコール番号が1でないならば分岐*/
    jsr GETSTRING       /*GETSTRING を呼び出す*/
    bra SYSTEM_CALL_FINISH

SYSTEM_CALL2:
    cmp.l   #2,%d0
    bne SYSTEM_CALL3        /*システムコール番号が 2 でないならば分岐*/
    jsr PUTSTRING       /*PUTSTRING を呼び出す*/
    bra SYSTEM_CALL_FINISH

SYSTEM_CALL3:
    cmp.l   #3,%d0
    bne SYSTEM_CALL4        /*システムコール番号が 3 でないならば分岐*/
    jsr RESET_TIMER     /*RESET_TIMER を呼び出す*/
    bra SYSTEM_CALL_FINISH

SYSTEM_CALL4:
    cmp.l   #4,%d0
    bne SYSTEM_CALL_FINISH  /*システムコール番号が 4 でないならば分岐*/
```

```
    jsr SET_TIMER       /*SET_TIMER を呼び出す*/

SYSTEM_CALL_FINISH:
    movem.l (%sp)+,%d1-%d3      /*レジスタの復帰*/
    rte

...

skipmt:
        rts
.end
```

ここでは，trap #0 の割り込み処理エントリの先頭にシステムコール番号 5 の判定を入れ，一致するとタイマ割り込み処理に jmp するように記述している．ここではタイマ割り込み処理を仮に skipmt とし，後方に何も処理を行わず rts で返ってくるという空の処理を施し，コンパイルエラーが出ないようにしている．この中身についてはテーマ 2 において実装を行う部分である．

4　プログラムにおいて注意したこと

4.1　成功判定とループ実行

入出力装置とプログラムの処理速度が一致しない場合，文字の読み取りや書き出しに失敗し，システムコール GETSTRING や PUTSTRING が正常に動作しないことがある．本プログラムでは，inbyte()および outbyte()において，失敗時に再試行を行う仕組みを導入した．具体的には，戻り値が格納されるレジスタ%d0 を確認し，成功するまでループでシステムコールを繰り返すようにしている．

4.2　outbyte()における引数

m68k-elf-gcc において，関数呼び出しの際の引数は右から左へ順に評価されてスタックに積まれる．そしてその後，復帰のための PC が積まれる．さらに，サブルーチン内でレジスタを退避させた場合，更にその上に積まれるような構造になっている．このため，outbyte()で引数を取ってきたい場合，現在の%SP の値をそのまま使うことはできず，欲しい引数の番地を計算し，%SP に加算する必要がある．仮に引数が仮に引数がアルファベット小文字の a であった場合のスタックの様子を図に示す．

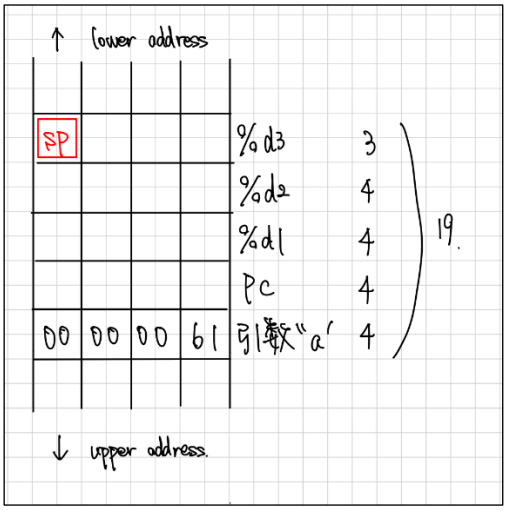

図 1：outbyte()実行時のスタックの状況

ここでは%SP に即値 19 を足すことによって，char ‘a’ = 0x00000061 のうち下 1 バイトだけを得られるということになる．このように，スタック構造を正確に理解し，計算によって引数のアドレスを取得することが重要であった．また，退避させたレジスタの数も考慮に入れなければ引数の操作が正確に行えないことにも注意を払って実装を行った．

4.3 中難度 Option: C のローカル変数領域に相当する作業領域の確保

この関数の実装を行う際, link.w %a6, #-4 命令は，現在のスタックポインタ(%sp)を基に，4 バイト分のローカル作業領域を確保するものであった．そのため入力されたものをアドレス計算の後にスタックに格納する必要があったが，-1(%a6)のアドレス操作は，作業領域サイズ（link で確保したバイト数）に依存するため，正しいサイズ設定を心がけ，バイトサイズで操作を行った．また，スタックポインタが乱れてプログラムが正しく動作しなくなることを防ぐために, unlk %a6 命令で作業領域の解放行うことを注意した．

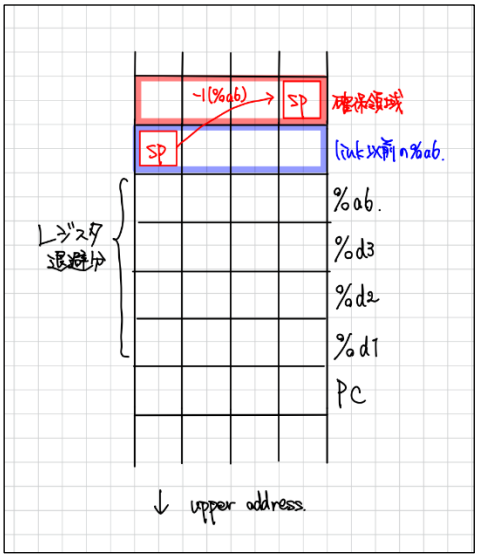

図 2 ：link 命令実行時のスタックの状況

5　プログラムに発生した問題とその原因，解決方法

5.1　outbyte()における引数

outbyte() の作成においては，上記の注意点で示したようにスタックへのデータの積まれ方，また対比させるレジスタの数を考慮したアドレス計算が必要であった．しかし，はじめはこのことに対する理解が不足しており，うまく関数を動かすことができなかった．その中で option 課題へのチャレンジやアドレスを 1 つずつずらしての地道な試行，また班員との相談などを通して解決することができた．

5.2　中難度 Option: C のローカル変数領域に相当する作業領域の確保

これも上記の注意点で示したものであるが，ローカル領域の仕様にチャレンジしている際に，アドレス操作の正しいサイズ設定に気づけておらず，関数がうまく動かない事態に陥った．これも一緒の課題にチャレンジしている班員とお互いのコードを検証し合う中で間違いに気づき，解決することができた．

6　考察

本実験を通じて，アセンブリ言語と C 言語の連携や，ハードウェア操作を習得することができた．特に，GETSTRING や PUTSTRING を利用したシステムコールの実装を通じて，標準入出力ライブラリの構造や，レジスタ操作，スタックフレームの管理方法を深く理解することができた．また，アセンブリ言語ではデータサイズやレジスタの配置を厳密に考慮する必要があり，その難しさを体感する一方で，効率的なメモリ操作が可能であることも

確認できた.

一方で, C 言語の高級ライブラリが, これらの低水準操作を抽象化し, 開発効率を向上させている理由も明確になった. 本実験の方法を応用すれば、, 他の C ライブラリ関数をターゲットシステム上で実現することも可能であり, さらには異なるプログラミング言語の移植や新しい OS の開発への応用も見込まれると考えられる. また, プログラムの分割コンパイルやモジュール化の重要性を再認識し, 大規模プログラムの保守性や拡張性を向上させる手法を学ぶことができたため, 今後のマルチタスクカーネルや応用プログラムの開発において大いに役立つと考える.

7　テーマ 2 の準備としての思考実験

ここでは, テーマ 2 の準備としての思考実験を行う. テーマ 2 において作成するセマフォとタイマ切り替えによるマルチタスク実行環境において, 複数のユーザタスクを実行した場合の各キューやセマフォの状態について考察する.

7.1　思考実験の概要と前提条件

今回の思考実験では以下の 2 点を満たすようにユーザタスクを作成を行う.

- セマフォを 2 つ以上使う
- 各セマフォ, ready キューの全てにおいて, 2 つ以上のタスクが入るタイミングが初期状態以外に存在する.

また, 本実験では実験遂行上の都合により, タイマ割り込みは 1 秒毎に発生し, タスクの切り替え及び P 命令, V 命令にかかる時間は限りなく短く, この間にタイマ割り込みが発生する可能性はないものとして考える.

7.2　作成したユーザタスク群とその説明

以下に, 作成したユーザタスク群のプログラムリストとその説明を示す.

```
#include <stdio.h>
#define N 1000 // hypothetical time unit

void task_1(void);
void task_2(void);
void task_3(void);
void task_4(void);
```

```c
int main(void)
{
    init_kernel(); // initialize the kernel
    set_task(task_1);
    set_task(task_2);
    set_task(task_3);
    set_task(task_4);
    begin_sch(); // start the scheduler
    return 0;
}

void task_1(void)
{
    while (1)
    {
        P(0);
        for (int i = 0; i < N; i++)
        {
            // loop for 1 second
        }
        V(0);
        P(1);
        for (int i = 0; i < N; i++)
        {
            // loop for 1 second
        }
        V(1);
    }
}

void task_2(void)
{
    while (1)
    {
        P(0);
        for (int i = 0; i < N; i++)
```

```
        {
            // loop for 1 second
        }
        V(0);
    }
}

void task_3(void)
{
    while (1)
    {
        P(1);
        for (int i = 0; i < N; i++)
        {
            // loop for 1 second
        }
        V(1);
    }
}

void task_4(void)
{
    while (1)
    {
        for (int i = 0; i < N * 3; i++)
        {
            // loop for 3 seconds
        }
    }
}
```

このプログラムでは, task_1 はセマフォ 0,1 の両方, task_2 はセマフォ 0 のみ, task_3 はセマフォ 1 のみを使い, それぞれ 1 秒間かかる処理, そして task_4 はセマフォを使わずに独立に 3 秒間かかる処理を記述している.

7.3 実行結果

上記の条件において，プログラムの実行を行う際の動作遷移表とその説明を以下に示す．

表 1：思考実験プログラムの動作遷移表

Events	current_task	ready	semaphore[0]		semaphore[1]	
			count	task_list	count	task_list
Initialize	-	1,2,3,4	1	-	1	-
begin_sch	1	2,3,4	1	-	1	-
P(0)[task_1]	1	2,3,4	0	-	1	-
timer before V(0)[task_1]	2	3,4,1	0	-	1	-
P(0)[task_2]	-	-	-1	2	1	-
P(1)[task_3]	3	4,1	-1	2	0	-
task_4	4	1,3	-1	2	0	-
V(0)[task_1]	1	3,2,4	0	-	0	-
P(1)[task_1]	-	-	0	-	-1	1
V(1)[task_3]	3	2,4,1	0	-	0	-
V(0)[task_2]	2	4,1	1	-	0	-
task_4	4	1	1	-	0	-
V(1)[task_1]	1	4	1	-	1	-
task_4	4	-	1	-	1	-
…						

これを見ると，task_2 及びセマフォ 0 の処理を終えてセマフォ 1 を利用しようとする task_1 に関して，セマフォが利用されていてロックされているために待ち行列に入っている時間があることが確認できる．また，task_4 はセマフォを利用しない独立した 3 秒間の処理を行うために， 1 秒毎のタイマ割り込みに合わせて 3 回に分けて実行されていることが確認できる．